Beyond Binary Classification

AW

Lecture Overview

1. **Loss/Gradient of Basic Neurons**
2. **DP: pre/post activation vars**
3. **Vectorization of Backpropagation**
4. **Batches and Gradient**
5. **Multiple Classes and Perceptron**
6. **Softmax**
7. **Feedforward Neural Nets**

Loss Functions – Birds Eye View

- Each Feedforward (FF-)net (including individual neurons) defines parametric model which gives a conditional distribution $p_{model}(y\,|\vec{x};\vec{\theta}\,)$ where $\vec{x}$ is our input and $\vec{\theta}$ is a vector of tunable (learnable) parameters of the model
- Our data is drawn from a general population distributed with some unknown distribution that we try to learn; we want to compare these distributions. Dissimilarity between the empirical distribution p_{data} defined by the training set and the model distribution p_{model} is often measured by the KL divergence:

$$D_{KL}\,(p_{data}, p_{model}) = \mathbb{E}_{x\sim p_{data}}\,[\log\big(p_{data}(x)\big) - \log p_{model}\,(x)]$$

- The first term under the expectation does not depend on training (is a function of data generation). Which means that we only need to maximize the second term $\mathbb{E}_{x\sim p_{data}}\,[\log p_{model}\,(x)]$ –cross entropy between distributions
- The latter coincides with definition of maximum likelihood which we see as an attempt to make the model distribution p_{model} match the empirical distribution p_{data}

Loss Functions – Birds Eye View (cont.)

- We want to estimate $\mathbb{E}_{(\vec{x},y)\sim p_{data}} p_{model}(y\,|\vec{x};\vec{\theta}\,)$ – i.e. cross-entropy between the training data and the model distribution.
- Of course we are looking for maximum likelihood estimate, so we train to have $\vec{\theta}_0 = \arg\max_{\vec{\theta}} \mathbb{E}_{(\vec{x},y)\sim p_{data}} p_{model}(y\,|\vec{x};\vec{\theta}\,)$
- For example if $p_{model}(y\,|\vec{x};\vec{\theta}\,) = \mathcal{N}(y; f(\vec{x},\vec{\theta}), 1)$ where f is a function computed by a network, i.e. $\hat{y} = f(\vec{x},\vec{\theta})$ then

$$\log p(y_i \mid x_i;\ \theta) = \frac{1}{2}\left(-\log\sigma - \log 2\pi - \frac{\|\hat{y}_i - y_i\|^2}{\sigma}\right)$$

so since $\sigma = 1$ we have

$$\mathbb{E}_{(\vec{x},y)\sim p_{data}} p_{model}(y\,|\vec{x};\vec{\theta}\,) = \frac{1}{2}\mathbb{E}_{(\vec{x},y)\sim p_{data}}\|y - \hat{y}\| + const$$

where constant does not depend on model parameters

Loss Functions – Taxonomy

- There are 3 types of loss functions for 3 different tasks: regression, binary classification and multiclass (categorical) classifications
- Regression:
  - if we assume Normal distribution of error we get SSE (SE on individual instance)
  - If we assume that $p_{model}(y\,|\vec{x};\vec{\theta}\,)$ I distributed Laplace $\mathcal{L}(y_i, \vec{x}_i, b) = \frac{1}{2b}\exp\left(-\frac{|y-\hat{y}|}{b}\right)$ with location parameter $b = 1$ the we get absolute error $|y - \hat{y}|$, etc.
- Binary classification max likelihood yield

$$p\log\hat{p} + (1-p)\log(1-\hat{p})$$

  i.e. probability of correct classification as we got for sigmoid
- We'll see later that multiclass classification likelihood yields

$$-\log\hat{y}_{oc}$$

  The loss that we'll have for softmax

Atomic Neurons Loss Functions Summary

Neuron Type	Loss
Linear activation (regression)	$\frac{1}{2}(y-\hat{y})^2$ (squared error – cross entropy of $\mathcal{N}(y:\hat{y},1)$)
ReLU	$\frac{1}{2}(y-\hat{y})^2$ (squared error – cross entropy of $\mathcal{N}(y:\hat{y},1)$)
Linear activation (regression on binary targets)	$\log(1+\exp(-y\cdot\hat{y}))$ (same as above but for binary targets)
Sigmoid (	$-\log\left\lvert\frac{y_i}{2}-0.5+\hat{y}_i\right\rvert$ (- log probability of correct prediction for Bernouli)
SVM	$\max(0,1-y\cdot\hat{y})$ (hinge loss)
Softmax (**later)**	$-\log y_r$ (cross entropy with y_r probability of prediction of correct class

Gradients Summary (again)

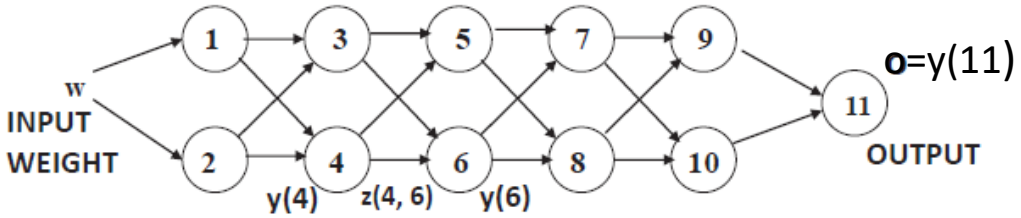
Activation function $o = \Phi(x)$	Gradient $o'_x = \left(\Phi(x)\right)'_x$
Linear (regression) $\Phi(x) = x$	$(x)'_x = 1$
Linear regression on binary targets $\Phi(x) = \text{sign}(x)$	$(\text{sign}(x))'_x = \begin{cases} 0 \text{ everywhere except } x = 0 \\ undefined \text{ at } x = 0 \end{cases}$
Sigmoid $\Phi(x) = \frac{1}{1-\exp(-x)}$	$\left(\frac{1}{1-\exp(-x)}\right)'_x = \frac{\exp(-x)}{(1-\exp(-x))^2} = \Phi(x)(1\text{-}\Phi(x))$
ReLU $\Phi(x) = \max(0, x)$	$(\max(0,x))'_x = \begin{cases} 1 \; if \; x > 0 \\ 0 \; otherwise \end{cases}$
SVM as in linear on binary targets	$(\text{sign}(x))'_x = \begin{cases} 0 \text{ everywhere except } x = 0 \\ undefined \text{ at } x = 0 \end{cases}$
Softmax $[\Phi(x)]_i = \frac{\exp(v_i)}{\sum_{j=1}^{n} \exp(v_j)}$ **(later)**	$\frac{\partial [\Phi]_i}{\partial \, v_j} = \begin{cases} [\Phi(x)]_i(1 - [\Phi(x)]_i) \; if \; i = j \\ -[\Phi(x)]_j \cdot [\Phi(x)]_i \end{cases}$

Lecture Overview

1. Loss/Gradient of Basic Neurons
2. DP: pre/post activation vars
3. Vectorization of Backpropagation
4. Batches and Gradient
5. Multiple Classes and Perceptron
6. Softmax
7. Feedforward Neural Nets

Recall Backpropagation

- We had directed acyclic graph (DAG) of operations as here
- We needed to run it backwards to compute gradient taking derivatives of operation in nodes along incoming edges



For example, on the figure derivative of the output of node 6 y(6) must have been with respect to its input – i.e. output y(4) of node 4. So $z(4,6)$ is in fact the partial derivative $z(4,6) = \frac{\partial y(6)}{\partial y(4)}$.

- For output $y(n)$ we computed $\nabla_{\boldsymbol{Y}} L = \frac{dL}{do} \cdot \nabla_{\boldsymbol{Y}} o$ where **Y** is tensor of variables and $o = y(n)$, so $\nabla_{\boldsymbol{Y}} y(n) = \mathbb{J}_Y \left(\overrightarrow{\mathbb{A}\left(y(n)\right)}\right)^T \nabla_{\overrightarrow{\mathbb{A}\left(y(n)\right)}} y(n)$ with the goal to compute $\nabla_{\vec{w}} o = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in \mathrm{P}} z(i,j)$ where $\vec{w}$ is a vector of all weights

  In other words for each variable we computed derivative along all paths from the source to it with the goal to have derivatives along all paths to output

Dynamic Programming and DAGs

- Dynamic programming used extensively in directed acyclic graphs (DAGs).
- Typical: Exponentially aggregative path-centric functions between source-sink pairs.
- General approach: Starts at either the source or sink and recursively computes the relevant function over paths of increasing length by reusing intermediate computations.
- In our backward propagation algorithm we have a path-centric function computed for the graph:

$$\nabla_{\boldsymbol{Y}} L = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in \mathrm{P}} z(i,j)$$

- Backwards direction makes more sense here because we have to compute derivative of output (sink) with respect to all variables in early layers.

The Idea Of Algorithm Introduced Earlier

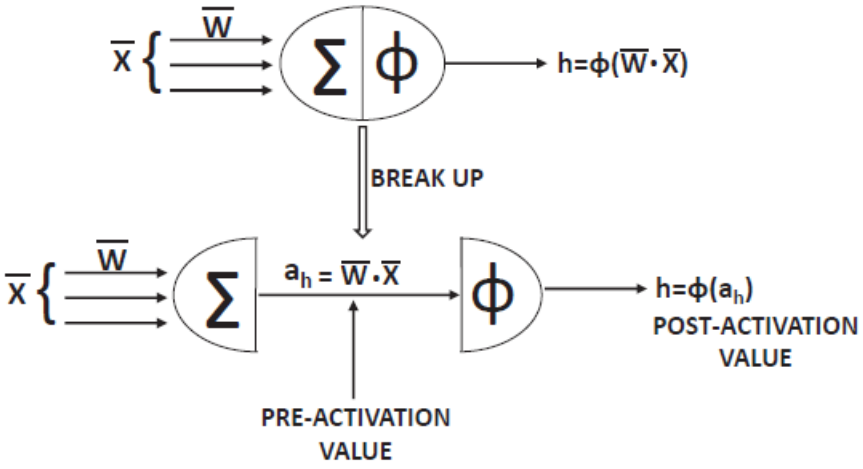
- Suppose for a given DAG $G(V, E)$ we need to compute a path-wise function $\mathrm{S}(b, o) = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in \mathrm{P}} z(i, j)$ where b is a source (if we need a vector of source we just introduce fake single source), o is a sink of a DAG and z is a function defined on arcs of the DAG
- In term of recurrence this can be re-written as

$$\mathrm{S}(b, o) = \sum_{b:(b,i) \in E} z(b, i) S(i, o)$$

  In fact the recurrence will hold if we want to compute any value for $S(j, o)$, just substitute j instead of b
- Dynamic programming approach will be to initialize $S(a, a)$ (for example take $S(o, o) = 1$), and recursively compute $S(i, o) = \sum_{j \in A(i)} S(j, o) z(j, i)$ where $A(i)$ is the set $\{j | (j, i) \in \mathrm{E}\}$ of ancestor nodes of i in partial order determined by DAG

Terms in Which We Compute $\nabla_{\boldsymbol{Y}} L$

- Our decoupled neurons in computation graph can be treated as here:



So after dot product we have pre-activation variable $a_h = \vec{w} \cdot \vec{x}$ after activation function applied we have post activation variable $h = \Phi(a_h)$

Recurrence Relation for Gradient

- To be consistent with the book denotes $S(i,j)$ of dynamic programming algorithm by δ_{ij} and define $\delta_{io} = \frac{\partial L}{a_h(i)}$ (plays the role of $S(i,o)$ in recurrence)
- Then $\delta_{io} = \frac{\partial L}{a_h(i)} = \sum_{i:(i,j)\in E} \frac{\partial L}{\partial a_h(j)} \cdot \frac{\partial a_{h(j)}}{\partial a_{h(i)}} = \sum_{i:(i,j)\in E} \delta_{jo} \cdot \frac{\partial a_{h(j)}}{\partial a_{h(i)}}$
- Notice that $a_{h(j)} = \sum_{k:(k,j)\in E} w_{kj} \Phi_k(a_h(k))$ so

$$\frac{\partial a_{h(j)}}{\partial a_{h(i)}} = \frac{\partial\left(w_{ij}\Phi_i(a_h(i))\right)}{\partial a_h(i)} = w_{ij}\frac{\partial \Phi_i(a_h(i))}{\partial a_h(i)} = w_{ij}\Phi_i'(a_h(i)) \text{ hence}$$

$$\delta_{io} = \Phi_i'(a_h(i)) \sum_{i:(i,j)\in E} w_{ij}\delta_{jo}$$

- Thus we have a recurrence relation for dynamic programming that give the following algorithm

Dynamic Programming Training Algorithm

1. Use a forward-pass to compute the values of all hidden units, output o, and loss L for a particular input-output pattern $(\vec{x}, y)$.
2. Initialize $\frac{\partial L}{\partial a_o} = \delta(o, o)$ to $\frac{\partial L}{\partial o}\Phi'(a_o)$
3. Use the recurrence $\delta_{io} = \Phi_i'(a_h(i)) \sum_{i:(i,j)\in E} w_{ij}\delta_{jo}$ to compute each $\delta(i, o)$ in the backwards direction for all nodes
4. After each computation for a node, compute the gradients with respect to incident weights as follows:

$$\frac{\partial L}{\partial w_{ij}} = \delta(j, o) \cdot h(i)$$

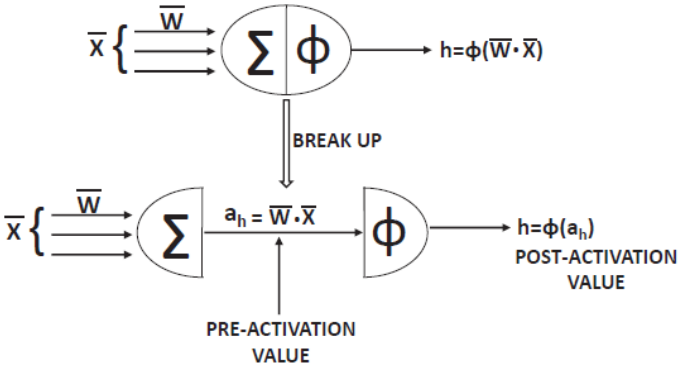
   The partial derivatives with respect to incident biases are computed substituting $+1$ instead of $h(i)$ (because bias neurons are always activated at a value of $+1$)
5. Use the computed partial derivatives of loss function wrt weights to update weights

Lecture Overview

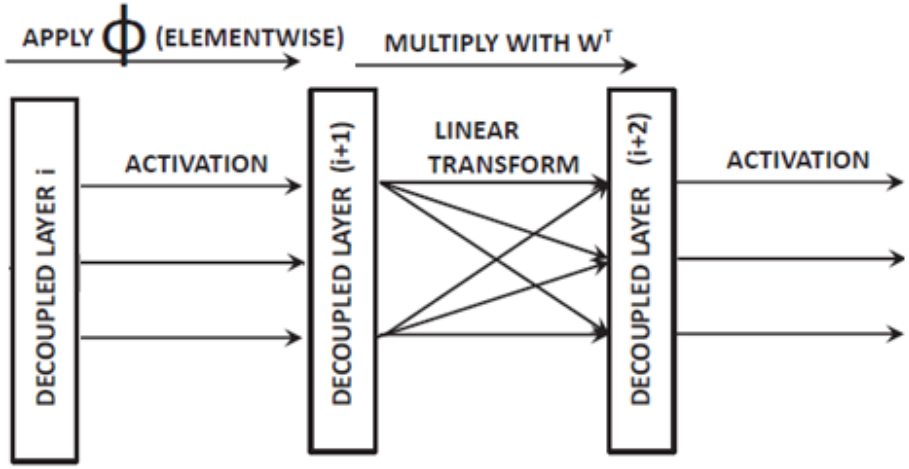
1. Loss/Gradient of Basic Neurons
2. DP: pre/post activation vars
3. **Vectorization of Backpropagation**
4. Batches and Gradient
5. Multiple Classes and Perceptron
6. Softmax
7. Feedforward Neural Nets

Decoupled View of a Neuron (again)

- Recall that we can view each neuron as an input layer and activation layer:



- So decouple neuron layer (split each into two). We now have:



- Notice that we renumbered layers: $i-1 \rightarrow i$ and $i \nearrow \begin{matrix} i+2 \\ i+1 \end{matrix}$

Vector Form of the Recurrence

- Let layer l_i consists of k neurons with activation functions $\Phi_1^{(i)}, \dots, \Phi_k^{(i)}$ and each neuron is connected to output of n neurons with outputs $h_1^{(i-1)}, \dots, h_n^{(i-1)}$ of layer l_{i-1}. Denote also activation value $a_h\left(j^{(i-1)}\right)$ by $z_{i-1,j}$
- Then for neuron with output $h_j^{(i-1)}$ consider our recurrence equation for computing g gradient :

$$\delta_{jo}^{(i-1)} = \left[\Phi_j^{(i-1)}\left(z_{i-1,j}\right)\right]' \sum_{s\in\{1,\dots,n\}} w_{js}^{i-1,i}\delta_{so}^{i}$$

$$= \begin{pmatrix} \left[\Phi_j^{(i-1)}\left(z_{i-1,j}\right)\right]' \\ \vdots \\ \left[\Phi_j^{(i-1)}\left(z_{i-1,j}\right)\right]' \end{pmatrix} \cdot \left(\begin{pmatrix} w_{j1}^{i-1,i} \\ \vdots \\ w_{jn}^{i-1,i} \end{pmatrix} \cdot \begin{pmatrix} \delta_{1o}^{i} \\ \vdots \\ \delta_{no}^{i} \end{pmatrix} \right)$$

Vector Recurrence in Decoupled Form

We have recurrence written as

$$\underbrace{\delta_{jo}^{(i)}}_{g_{ij}} = \begin{pmatrix} \left[\Phi_j^{(i)}(z_{i,j})\right]' \\ \vdots \\ \left[\Phi_j^{(i)}(z_{i,j})\right]' \end{pmatrix} \cdot \underbrace{\left(\begin{pmatrix} w_{j1}^{i+1,i+2} \\ \vdots \\ w_{jn}^{i+1,i+2} \end{pmatrix} \cdot \overbrace{\begin{pmatrix} \delta_{1o}^{i+2} \\ \vdots \\ \delta_{no}^{i+2} \end{pmatrix}}^{\vec{g}_{i+2}} \right)}_{\vec{g}_{i+1,j}}$$

- Denote $\vec{g}_{i+2} = \begin{pmatrix} \delta_{1o}^{i} \\ \vdots \\ \delta_{no}^{i} \end{pmatrix}$ - gradient of the next layer
- Denote $\vec{g}_{i+1,j} = \underbrace{\left(\begin{pmatrix} w_{j1}^{i+1,i+2} \\ \vdots \\ w_{jn}^{i+1,i+2} \end{pmatrix} \cdot \overbrace{\begin{pmatrix} \delta_{1o}^{i+2} \\ \vdots \\ \delta_{no}^{i+2} \end{pmatrix}}^{\vec{g}_{i+2}} \right)}_{\vec{g}_{i+1}}$ and $\vec{g}_{i+1} = \begin{pmatrix} g_{i+1,1} \\ \vdots \\ g_{i+1,k} \end{pmatrix}$

  so $\vec{g}_{i+1}$ is exactly the gradient vector of linear layer
- The gradient of layer i is $\vec{g}_i = \begin{pmatrix} g_{i1} \\ \vdots \\ g_{ik} \end{pmatrix}$ where g_{ij} is defined by recurrence equation

Matrix form of Forward Computation

Forward direction:

- To be consistent in numbering after decoupling let now $\vec{z}_{i+1}$ be the vector of output of activation layer and $\vec{z}_{i+2}$ be output of dot product layer.
- Then we have $\vec{z}_i$ vector going forward

$$\vec{z}_{i+1=}\overrightarrow{\Phi} * (\vec{z}_i)$$

where $\overrightarrow{\Phi} * (\vec{z}_i) = \begin{pmatrix} \Phi_1(z_{i1}) \\ \vdots \\ \Phi_k(z_{ik}) \end{pmatrix}$ and $\vec{z}_{i+2} = \left(W^{i+1,i+2}\right)^T \vec{z}_{i+1}$

- if $\Phi_{ij} = \Phi_{ik}$ for all j, k we have $\vec{z}_{i+1=}\Phi(\vec{z}_i)$ and

Backpropagation in Matrix Form

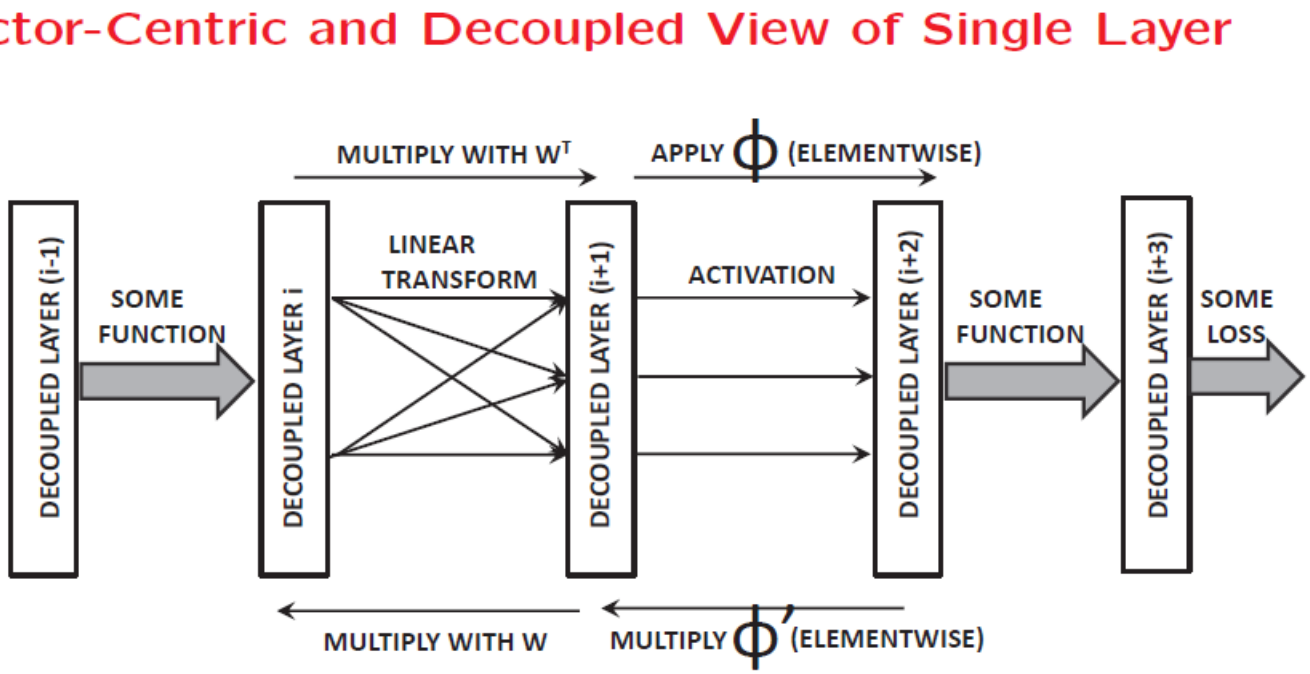
- The in the forward propagation w_{kj}^{f} is the weight from neuron j at level i to neuron k at level $i+1$ in our denotations in backward propagation w_{jk}^{b} is a weight from neuron j at level i to neuron k at level $i+1$, so $W^{i,i+1} = \left(W^{i+1,i}\right)^{T}$ and therefore $\vec{g}_{i+1} = W^{i+1,i+2}\vec{g}_{i+2}$
- In most cases all activation functions $\Phi_{k}^{(i)}$ in one layer are the same function Φ. In this case let $\Phi'(\vec{z}_i)$ denote matrix

$$\begin{pmatrix} \Phi'(z_{i1}) & 0 & 0 \dots & 0 \\ 0 & \Phi'(z_{i2}) & 0 \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 \dots & 0 \dots & 0 & \Phi'(z_{ik}) \end{pmatrix}, \text{ then } \vec{g}_i = \Phi'(\vec{z}_i)\vec{g}_{i+1};$$

  - In most general layered case when all activation functions are different and take many (possibly all inputs from the previous) layer we have formula with which we started:

$$\vec{g}_i = (\mathbb{J}_{\vec{y}_i}\Phi)(\vec{z}_i)\vec{g}_{i+1};$$

Training in Matrix Form



- Note that linear matrix multiplication and activation function are separate layers
- We can use the vector-to-vector chain rule to backpropagate on a single path

Standard cases

- In standard cases no need to do calculation! Can just use precomputed values
- Most come form the gradient of activation functions table but here they are again:

Function	Forward	Backward
Linear	$\overline{z}_{i+1} = W^T \overline{z}_i$	$\overline{g}_i = W \overline{g}_{i+1}$
Sigmoid	$\overline{z}_{i+1} = \text{sigmoid}(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (1 - \overline{z}_{i+1})$
Tanh	$\overline{z}_{i+1} = \tanh(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot (1 - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	Set to ± 1 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)

Lecture Overview

1. Loss/Gradient of Basic Neurons
2. DP: pre/post activation vars
3. Vectorization of Backpropagation
4. Batches and Gradient
5. Multiple Classes and Perceptron
6. Softmax
7. Feedforward Neural Nets

Batch Learning

- What we have used when explaining backpropagation was a training set of size 1. This works for online learning when we compute adjustments as soon as the example arrives, which is why it is called *online learning*
- The gradients can also be computed with respect to multiple training instances at one time, and these multiple instances are referred to a *batch*
    - What we did in the program for the linear regression was either full training set (full batch) or we divided the training set into smaller subsets (small batch) of training examples

Stochastic Batch Learning

- When using a randomly selected subset of from the training set of the size n, we say that we use *stochastic gradient descent* or *minibatch learning* (with batch size n).
  - This type of gradient descent is referred to as *stochastic* because one cycles through the points in some random order.
  - interesting property of stochastic gradient descent is that even though it might not perform as well on the training data (compared to online gradient descent), it often performs comparably on test data and train data. Intuitively this is because sampling represents general population ordering better than a snapshot (order of training data)
- Training on one batch is called *iteration*
- Training on the full data set once is called *epoch*

Loss Function in Batch Learning

- In single/online learning we used Squared Errors (SE) for the loss
- In batch case its variant in use is the *mean squared error* (MSE).
  - Since we used the batch we need to rewrite the cost function as the average of the cost functions $SE(x)$ for individual training samples *x*, and we therefore for a batch $b = \{(\vec{x}_1, y_1), \dots, (\vec{x}_n, y_n)\}$ of size n we define

$$MSE(b) = \frac{1}{n}\sum_{i=1}^{n} SE(\vec{x}_i, y_i)$$

  - In the program we used batches and MSE
- In general if L_i is the loss on one instance of data we can define on a batch as either total loss on a batch $L_t = \sum_{i=i}^{n} L_i$ or an average loss on a batch $L_a = \frac{1}{n}\sum_{i=i}^{n} L_i$

Batches - update

- If loss is defined based on sum of losses of instances in a batch then from the sum property of gradient we have that total loss $\nabla_{\vec{w}} L_t = \sum \nabla_{\vec{w}} L_i$ (SSE=Sum of SE) and average loss $\nabla_{\vec{w}} L_a = \frac{1}{n} \sum \nabla_{\vec{w}} L_i$
- Since the update on individual instance is given as

$$\vec{w}' = \vec{w} + \alpha(-\nabla_{\vec{w}} L_i)$$

it is natural that if we define the updates for total loss or average loss respectively as $\vec{w}' = \vec{w} + \alpha(-\sum \nabla_{\vec{w}} L_i)$ and

$$\vec{w}' = \vec{w} + \frac{1}{n} \alpha(-\sum \nabla_{\vec{w}} L_i)$$

- In all 3 cases (individual, butch total, batch average) we are going to get the same result of gradient descent

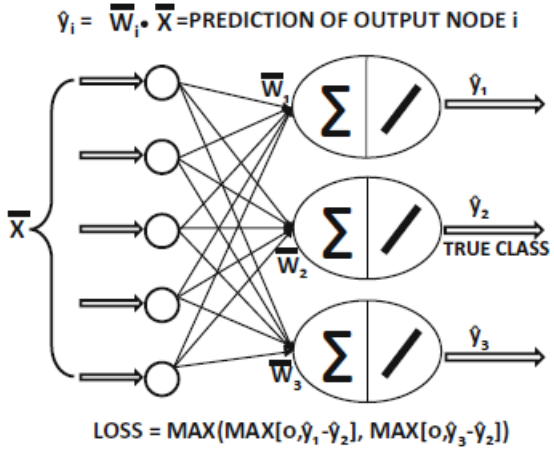
Lecture Overview

1. Loss/Gradient of Basic Neurons
2. DP: pre/post activation vars
3. Vectorization of Backpropagation
4. Batches and Gradient
5. Multiple Classes and Perceptron
6. Softmax
7. Feedforward Neural Nets

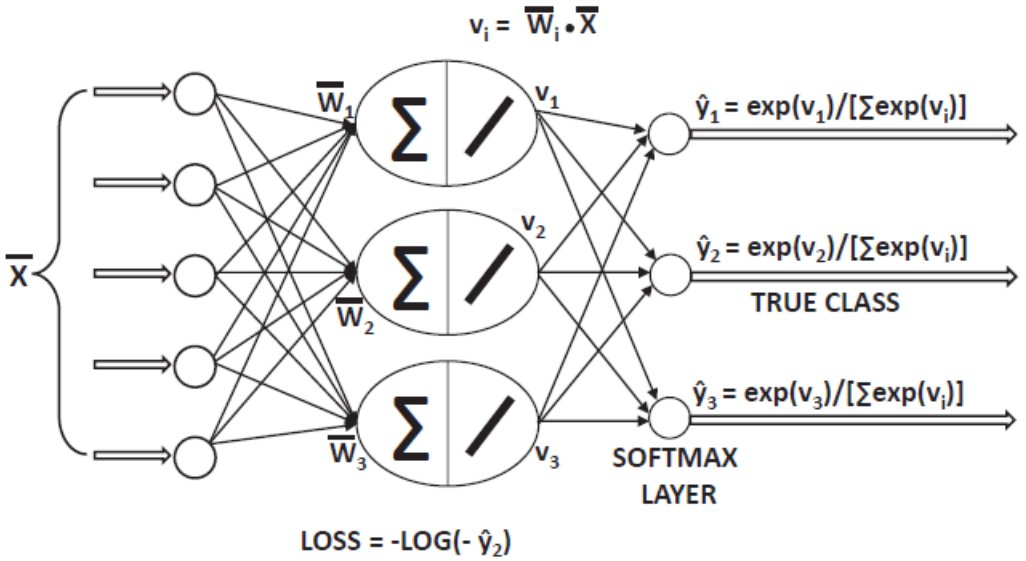
Binary Classes versus Multiple Classes

- All classification models discussed so far (perceptron, SVM, Logistic regression) are only for binary classification in which the class label is drawn from $\{-1, +1\}$.
- More often than not we need multiple classes:
  - Image classification: predicting the category of an image (e.g., *car*, *person, landscape*)
  - *Language models:* Predict the next word in a sentence.
- Previous models are naturally designed to predict two classes, can we modify them?

Two Multiclass Models



Multiclass Perceptron



Softmax Neuron

Multiclass Perceptron

- Data: k-classes instance $(\vec{x}_i, c(i))$ contains a d-dimensional feature vector $\vec{x}_i$ and the class number $c(i) \in \{1, \ldots, k\}$ of its observed class.

The idea: find k different linear separators (vectors defining separation planes) $\vec{w}_1, \ldots, \vec{w}_k$ simultaneously so that the value of $\vec{w}_{c(i)} \cdot \vec{x}_i > \vec{w}_j \cdot \vec{x}_i$ for all $j \neq c(i), j \in \{1, \ldots, k\}$

- In other words we predicts a data instance X_i to belong to a class r with the largest value of $\vec{w}_r \cdot \vec{x}_i$, i.e.

$$c(i) = \arg \max_{r \in \{1,\ldots,k\}} \vec{w}_r \cdot x_i$$

- What is the loss function? It is 0 if class is determined correctly and it must be 'misclassification value' if $\max_{r \in \{1,\ldots,k\}} \{\vec{w}_r \cdot x_i\} \neq c(i)$ which is

$$\max\left(\max_{\substack{r \in \{1,\ldots,k\} \\ r \neq c(i)}} \{\vec{w}_r \cdot x_i - \vec{w}_{c(i)} \cdot \vec{x}_i\}, 0\right) \text{ or equivalently}$$

$$L_i = \max_{\substack{r \in \{1,\ldots,k\} \\ r \neq c(i)}} \{\max(\vec{w}_r \cdot \vec{x}_i - \vec{w}_{c(i)} \cdot \vec{x}_i, 0)\}$$

Update Rule for Multiclass Perceptron

If we denote by $\hat{y}_j^i$ the prediction value $\vec{w}_j \cdot \vec{x}_i$ of target variable for class j then we can write

$$L_i = \max_{\substack{r \in \{1,\ldots,k\} \\ r \neq c(i)}} \{\max(\hat{y}_r^i - \hat{y}_{c(i)}^i, 0)\}$$

Clearly taking partial gradients for every $\vec{w}$ we get that

$$\nabla_{\vec{w}_j} L_i = \begin{cases} -\vec{x}_i \text{ if } j = c(i) \\ \vec{x}_i \text{ if } j = \arg\max_{r \in \{1,\ldots,k\}} \vec{w}_r \cdot x_i \\ 0 \text{ for all other } j \in \{1, \ldots, k\} \end{cases}$$

So the update rule for each class j in multiclass perceptron is

$$\vec{w}_j' = \vec{w}_j + \begin{cases} \alpha\vec{x}_j \text{ if } j = c(i) \\ -\alpha\vec{x}_j \text{ if } j = \arg\max_{r \in \{1,\ldots,k\}} \vec{w}_r \cdot x_i \\ 0 \text{ for all other } j \in \{1, \ldots, k\} \end{cases}$$

Lecture Overview

1. Loss/Gradient of Basic Neurons
2. DP: pre/post activation vars
3. Vectorization of Backpropagation
4. Batches and Gradient
5. Multiple Classes and Perceptron
6. Softmax
7. Feedforward Neural Nets

Multinomial Logistic Regression

- Logistic regression produces probabilities of the two outcomes of a binary class.
- *Multinomial* logistic regression produces probabilities of multiple outcomes.
  - To produce probabilities of multiple classes activation function must output a vector of probabilities.
- The *softmax activation function* is vector-producing generalization of the sigmoid activation used in logistic regression.
- Multinomial logistic regression is also referred to as softmax classifier.

Model for Multinomial Logistic Regression

- Data: k-classes instance $(\vec{x}_i, c(i))$ contains a d-dimensional feature vector $\vec{x}_i$ and the class number $c(i) \in \{1, \dots, k\}$ of its observed class
- Suppose that there is another value of target variable $k+1$ such that space $\mathcal{P} = \bigcup_{\vec{x} \in \mathrm{X}} \left[(\vec{x}, y = k+1) \cup \bigcup_{i=1}^{k} (\vec{x}, y = i) \right]$ is complete probability space and for a given $\vec{x}$ complete set of events is $(\vec{x}, k+1) \cup \bigcup_{i=1}^{k} (\vec{x}, y = i)$
- We know that joint probability on pairs $(\vec{x}, t) \in \mathcal{P}$ conditioned on $\vec{x}$ is such that there is a linear relationship between feature variables and log-odds of two events: the event that $(\vec{x}, \mathrm{y} = i)$ and the event $(\vec{x}, y = k+1)$. Let probability p_i denote $p(y = i)$ and p_c denote $p(c)$. That is we know that for each i we have $\ln \frac{p_i}{p_{k+1}} = \vec{w}_i \cdot \vec{x}$.

Softmax Activation

- Model: k-classes instance $(\vec{x}_i, c(i))$ contains a d-dimensional feature vector $\vec{x}_i$ and the class number $c(i) \in \{1, \dots, k\}$ of its observed class such with probability distribution $p_i(\vec{x}) = p(\vec{x}, y = i)$ such that $\ln \frac{p_i}{p_{k+1}} = \vec{w}_i \cdot \vec{x}$ for $i \in \{1, \dots, k\}$
- Assuming $\ln \frac{p_i(\vec{x})}{p_{k+1}(\vec{x})} = \vec{w}_i \cdot \vec{x}$ we get $\sum_{i=1}^{k} \frac{p_i(\vec{x})}{p_{k+1}(\vec{x})} = \sum_{i=1}^{k} e^{\vec{w}_i \cdot \vec{x}}$.
- By law of complete probability $p_{k+1}(\vec{x}) + \sum_{i=1}^{k} p_i(\vec{x}) = 1$. Hence $\frac{1 - p_{k+1}(\vec{x})}{p_{k+1}(\vec{x})} = \frac{\sum_i^k p_i(\vec{x})}{p_{k+1}(\vec{x})} = \sum_{i=1}^{k} \frac{p_i(\vec{x})}{p_{k+1}(\vec{x})} = \sum_{i=1}^{k} e^{\vec{w}_i \cdot \vec{x}}$.
- Suppose $p_{k+1}(\vec{x})$ have equal odds of happening ($p_{k+1}(\vec{x}) = \frac{1}{2}$ and $\sum_{i=1}^{k} e^{\vec{w}_i \cdot \vec{x}} = \frac{1}{2}$) then $p_i = p_{k+1} e^{\vec{w}_i \cdot \vec{x}} = \frac{1}{2} \cdot \frac{e^{\vec{w}_i \cdot \vec{x}}}{\sum_{i=1}^{k} e^{\vec{w}_i \cdot \vec{x}}}$ so if we want to classify the feature vector x by the class that has $\max p_i$ among k-classes, i.e. we want to maximize $\frac{e^{\vec{w}_i \cdot \vec{x}}}{\sum_{i=1}^{k} e^{\vec{w}_i \cdot \vec{x}}}$
- Based on this model we define softmax activation function

$$\Phi(\vec{x}_i) = \arg \max_{j \in \{1, \dots, k\}} \left\{ \frac{e^{\vec{w}_j \cdot \vec{x}_i}}{\sum_{j=1}^{k} e^{\vec{w}_j \cdot \vec{x}_i}} \right\}$$

Loss Function for Softmax

- We used the negative logarithm of the probability of observed class in binary logistic regression.
- Natural generalization to multiple classes.
  - Cross-entropy loss: Negative logarithm of the probability of correct class:

$$L_i = -\ln\left(p(c(i)|\vec{x}_i)\right)$$

$$= -\ln\frac{e^{\vec{w}_{c(i)}\cdot\vec{x}_i}}{\sum_{j=1}^{k} e^{\vec{w}_j\cdot\vec{x}_i}}$$

$$= -\vec{w}_{c(i)}\cdot\vec{x}_i + \ln\left(\sum_{j=1}^{k} e^{\vec{w}_j\cdot\vec{x}_i}\right)$$

  - Probability distribution among incorrect classes has no effect!

Gradient for Softmax

- Using standard denotations of $\hat{y}_j^i$ the prediction value $\vec{w}_j \cdot \vec{x}_i$ of target variable for class j we can write

$$L_i = -\hat{y}_i^{c(i)} + \ln\left(\sum_{j=1}^{k} \exp \hat{y}_i^j\right)$$

- Softmax activation is used almost exclusively in output layer and (almost) always paired with cross-entropy loss.

- Therefore $\nabla_{\vec{w}_j} L_i = \begin{cases} -\vec{x}_i \left(1 - \frac{\exp(\vec{w}_{c(i)} \cdot \vec{x}_i)}{\sum_{j=1}^{k} \exp(\vec{w}_j \cdot \vec{x}_i)}\right) & \text{if } j = c(i) \\ \vec{x}_i \cdot \frac{\exp(\vec{w}_{c(i)} \cdot \vec{x}_i)}{\sum_{j=1}^{k} \exp(\vec{w}_j \cdot \vec{x}_i)} & \text{if } j \neq c(i) \end{cases}$

$$= \begin{cases} -\vec{x}_i \left(1 - p(c(i)|\vec{x}_i)\right) \text{ if } j = c(i) \\ x_i p(j|\vec{x}_i) \text{ if } j = c(i) \end{cases}$$

Update Rule for Softmax

- In the gradient each of the terms $[1 - P(c(i)|\vec{x}_i)]$ and
- $P(j|\vec{x}_i)$ where $j \neq c(i)$ is the probability of making a mistake for an instance with label $c(i)$ with respect to the predictions for the j^{th} class so it is natural that update rule will correct the mistake by corresponding expected value:

$$\vec{w}_j' = \vec{w}_j + \begin{cases} \alpha\vec{x}_i \left(1 - \frac{\exp(\vec{w}_{c(i)} \cdot \vec{x}_i)}{\sum_{j=1}^{k} \exp(\vec{w}_j \cdot \vec{x}_i)}\right) & \text{if } j = c(i) \\ -\alpha\vec{x}_i \cdot \frac{\exp(\vec{w}_{c(i)} \cdot \vec{x}_i)}{\sum_{j=1}^{k} \exp(\vec{w}_j \cdot \vec{x}_i)} & \text{if } j \neq c(i) \end{cases}$$

$$= \vec{w}_j + \begin{cases} \alpha\vec{x}_i\left(1 - p(c(i)|\vec{x}_i)\right) & \text{if } j = c(i) \\ -\alpha x_i p(j|\vec{x}_i) & \text{if } j = c(i) \end{cases}$$

Hierarchical Softmax

- Let our problem have very big number of classes
  - Often intext mining where the prediction is a target word
- Hard to solve because of inefficiency - learning is slow because need to have big number of separators
- Try *hierarchical softmax*: decompose the classification into hierarchy: group the classes into a binary tree-like structure, and then perform $\log_2(k)$ binary classifications from the root to the leaf for k-way classification.
- How to group? When no idea – at random. But grouping to 'close' classes usually improves performance
  - For words close could be
    - Semantically close (for example as given by WordNet)
    - High frequency of occurrence (binary tree by Huffman encoding)

Lecture Overview

1. Loss/Gradient of Basic Neurons
2. DP: pre/post activation vars
3. Vectorization of Backpropagation
4. Batches and Gradient
5. Multiple Classes and Perceptron
6. Softmax
7. Feedforward Neural Nets (again)

Feedforward Network Architecture

Recall that

- Most neural networks are organized into groups of units called layers.
- Most neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it.
- If each unit of the next layer has as input weighted outputs of all neurons of previous layer then these networks are called complete *feedforward networks*
- In this structure, the first layer is given by

$$\vec{h}^1 = \Phi^1\left((W^1)^T\vec{x} + \vec{b}^1\right)$$

where $\vec{x}$ is n-dimensional input vector, W^1 is $n \times m$ matrix of weights with column $\vec{w}_i$ being column of input weights of neuron i, Φ^1 is activation function applied to its vector argument component-wise, $\vec{b}^1$ is a bias vector, with b_i^1 bias of neuron i, and $\vec{h}^1$ is layer 1 output with h_i^1 being output of neuron i of layer 1.

- The following layers are defined similarly:

$$\vec{h}^{i+1} = \Phi^i\left((W^1)^T\vec{h}^i + \vec{b}^i\right)$$

Approximation Theorem

Universal approximation theorem (Hornik *et al.*, 1989; Cybenko, 1989; Leshno 1993). A feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function or ReLU) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.

- For all intensive purposes 'Borel measurable function' means here continuous function on a closed bounded subset
- Note that these networks are complete feedforward
- This means that a large enough feedforward network is able to *compute* any function.
- **This does not mean** that they are guaranteed that the training algorithm will be able to *learn* that function.

Reading

- 2.3.1-2.3.4
- 3-1
- 3.2